

# The itools XML infrastructure

J. David Ibez

europython 2005

`jdavid@itaapy.com`



# Introduction: Who is the speaker

- Python developer for 8 years. Zope developer for 6 years.
- CTO of Itaapy, a (French) company specialized on web applications development.
- Lead developer of the `itools` library, the *iKaaro* CMS, and the Localizer Zope product.



# Introduction: What is itools

A Python library, basically made of:

`itools.uri`

`itools.types`

`itools.resources`

`itools.handlers`

`itools.workflow`

`itools.catalog`

`itools.xml`

`itools.xhtml`

`itools.html`

`itools.gettext`

`itools.i18n`

And coming soon:

`itools.ical`

`itools.tmx`

`itools.xliff`



# Summary

## The Core:

- The Parser (`itools.xml.parser`)
- XML Namespaces (`itools.xml.namespaces`)
- XML Documents (`itools.xml.XML`)

## High-Level tools:

- The Simple Template Language (`itools.xml.stl`)
- XHTML Documents, the translation chain (`itools.xhtml`)

To close with directions for future developments.



# The Parser



# The Parser

At the bottom of the tool-set there is the XML parser.

There are a myriad of XML parsers available in the Python ecosystem, why yet another one?

The purpose of `itools.xml.parser` is to provide a better programming interface.

As a reference, we are going to see how the programming interface provided by `itools.xml.parser` differs from the one provided by `expat`.



# The Parser: expat

```
def start_element(name, attrs):
    print 'Start element:', name, attrs
def end_element(name):
    print 'End element:', name
def char_data(data):
    print 'Character data:', repr(data)

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse(data)
```



# The Parser: itools.xml

```
from itools.xml import parser

for event, value, line_number in parser.parse(data):
    if event == parser.START_ELEMENT:
        namespace, prefix, local_name = value
        print 'START ELEMENT:', local_name
    elif event == parser.END_ELEMENT:
        namespace, prefix, local_name = value
        print 'END ELEMENT:', local_name
    elif event == parser.TEXT:
        print 'TEXT', value
```



# The Parser

The main difference compared with `expat`, and with other parsers like `sax`, is that the code is all together within the same loop, instead of being dispersed across different handler methods.

This makes the code more compact, and easier to follow the control flow. It is also potentially faster, since we avoid many function calls, and the parsing variables are local.

# The Parser

The types of events currently supported are:

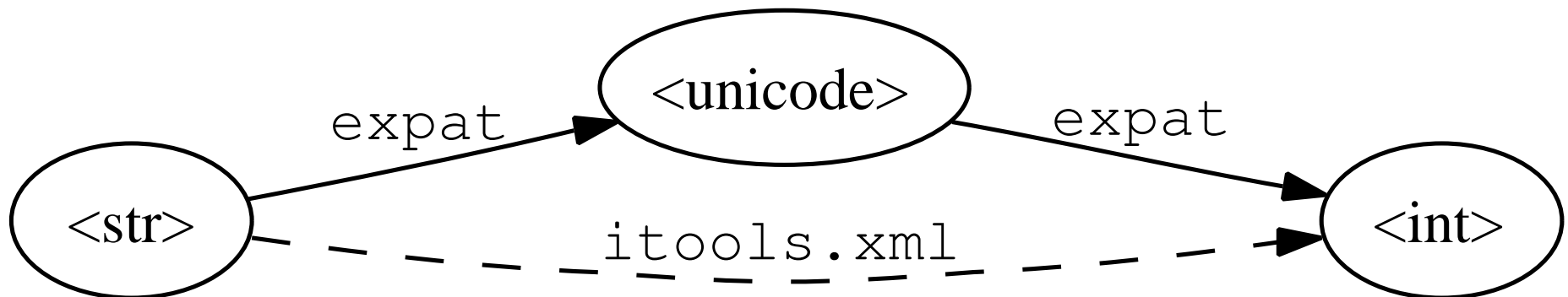
Event	Value
XML_DECLARATION	(xml version, encoding, standalone)
DOCUMENT_TYPE	(name, system id, public id, has internal subset)
START_ELEMENT	(namespace uri, prefix, local name)
END_ELEMENT	(namespace uri, prefix, local name)
ATTRIBUTE	(namespace uri, prefix, local name, value)
COMMENT	value
TEXT	value

# The Parser

The values returned for attributes, text and comment nodes are always byte strings. With the source encoding we will be able to de-serialize them.

This behavior differs from `expat`'s, where unicode strings are returned by default.

The purpose is to speed up de-serialization when we want to load values as something else. For instance:



# The Parser

Today `itools.xml.parser` is implemented as a wrapper around `expat`. This makes it relatively slow.

A native parser providing the same programming interface is on the works. This not only will improve the performance.

It also will open the door for other programming interface enhancements, like the ability to parse non well-formed documents (fragments for instance).



# XML Namespaces

# XML Namespaces

The main purpose of the support for XML namespaces provided by `itools.xml.namespaces` is to de-serialize values. For instance:

```

```

Here, the value for the attribute `title` should be loaded into memory as a Unicode string, while the `src` attribute should be loaded as a URI reference (a type provided by `itools.uri`), and the `border` value should be an integer.



# XML Namespaces

For example:

```
from itools.xml import parser, namespaces
from itools import xhtml

for event, value, line_number in parser.parse(data):
    if event == parser.ATTRIBUTE:
        ns_uri, prefix, local_name, value = value
        ns = namespaces.get_namespace(ns_uri)
        schema = ns.get_attribute_schema(local_name)
        value = schema['type'].decode(value)
```



# XML Namespaces

Basically what `itools.xml.namespaces` allows is to define schemas that describe attributes or elements for a given XML Namespace.

XML Documents, we will see them later, use the schema to get the type for elements and attributes. An information used to de-serialize values and build a rich element tree.



# XML Namespaces

Several XML namespace handlers are provided by `itools` out-of-the-box, including:

- The *default namespace*, used when the element or the attribute is not associated to any namespace.
- The special namespace “`xml`”, used for some core attributes (e.g. `xml:lang`).
- The special namespace “`xmlns`”, used for the namespace declarations.
- Dublin Core (`dc`).
- XHTML.

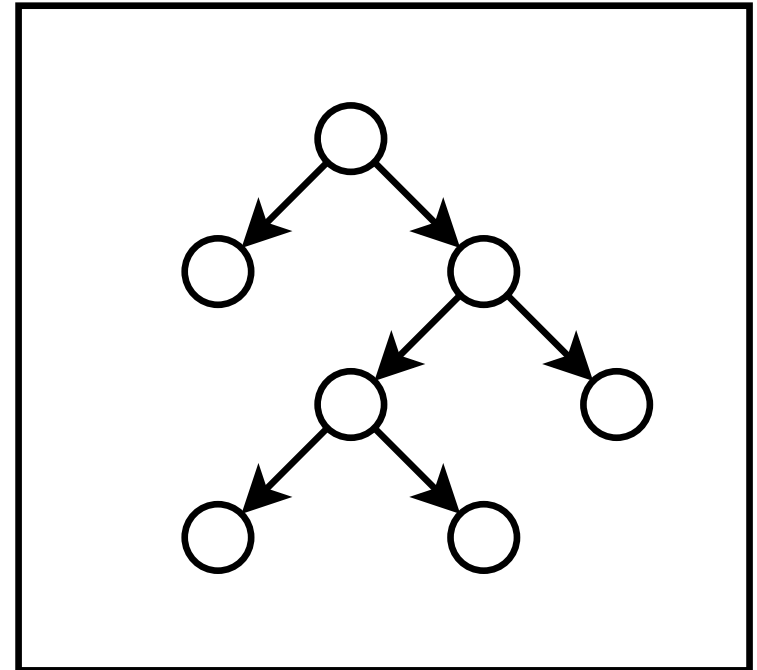
# XML Documents

# XML Documents

The *handler class*\* `itools.xml.XML.Document` represents an XML file as an element tree, a DOM-like data structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <head>
    <title>itools rocks!</title>
  </head>
  <body>
    <h1>itools rocks!</h1>
    <p>
      That's it, itools has changed the live of
      so many, bringing fun back to work, it
      has...
    </p>
    <a href="http://www.ikaaro.org">Click</a>
  </body>
</html>
```

Resource (an XML file)



The handler's state (an element tree)

# XML Documents

*\* handler class:*

A resource handler is a non-persistent object associated with a resource (a file or a folder), and responsible to manage it.

It represents the resource content with an on-memory data structure, and provides an API to work with it.

`XML.Document` is a file handler class for XML documents. Its instances are file handlers.

The resource-handler core is implemented by `itools.resources` and `itools.handlers`.



# XML Documents: API

The API includes:

- `get_root_element()`  
Returns the root element.
- `traverse()`  
A generator that traverses the XML tree in pre-order, and returns each time a node.
- `traverse2()`  
A more powerful version of `traverse`, it allows to control the traversal process.

# XML Documents: API

An example here:

```
>>> from itools.xml import XML
>>> from itools.xhtml import XHTML
>>>
>>> doc = XHTML.Document(title="Hello World")
>>> for node in doc.traverse():
...     if isinstance(node, XML.Element):
...         if node.name == 'title':
...             print node.get_content()
...             break
...
Hello World
```



# The Simple Template Language



# The Simple Template Language

There are a myriad of template languages, why yet another one?

Because none of the available solutions met our requirements:

- Clean separation between logic and presentation.
- Simple.
- Fast.



# The Simple Template Language

The namespace

```
{'title': 'Hello World',  
 'description': ...  
}
```

The template

```
<head>  
<title stl:content="title">  
  The title  
</title>
```

itools.xml.stl

```
<head>  
<title>Hello World</title>
```



# The Simple Template Language

There are always two sides:

- The template  
XML with STL elements and attributes. It represents the presentation side.
- The namespace  
Typically a dictionary or an object. Built from Python it represents the logic side.

The recommended development procedure is to write first the template, then the namespace, iterating over as needed.

# The Simple Template Language

The Simple Template Language is an XML namespace, hence it must be declared:

```
<html xmlns:stl="http://xml.itools.org/namespaces/stl">
```

# The Simple Template Language

Two elements:

- `<stl:block></stl:block>`
- `<stl:inline></stl:inline>`

Four attributes:

- `stl:content="expression"`
- `stl:attributes="name expression;  
                  name expression;  
                  ..."`
- `stl:if="expression"`
- `stl:repeat="name expression"`



# The Simple Template Language

There are two kinds of expressions:

Boolean expression : `[not] <path expression>`

Path expression : `<name>[ /<name> ]*`

# The Simple Template Language

Example. The template:

```
<h1 stl:content="title">The Title</h1>
```

```
<stl:block repeat="object objects">
```

```
  <a href="#" stl:attributes="href object/url"
```

```
    stl:content="object/title">Object's link</a>
```

```
  <br/>
```

```
</stl:block>
```



# The Simple Template Language

Example. The namespace:

```
from itools.handlers import get_handler

# Load the template
template = get_handler('template.xhtml')
# Build the namespace
namespace = {}
namespace['title'] = u'Fruits'
namespace['objects'] = [
    {'title': u'Orange', 'url': 'orange'},
    {'title': u'Apple', 'url': 'apple'}]
# Go!
print template.stl(namespace)
```



# The Simple Template Language

Have the goals been satisfied:

- Clean separation between logic and presentation?  
**YES.** Presentation is expressed on XHTML and STL, logic is written in Python.
- Simple?  
**YES.** The whole STL language can be learnt in few hours.
- Fast?  
**YES.** The simplicity of STL has made possible a simple and efficient implementation.

# XHTML Documents, the translation chain



# Translating XHTML Documents

The package `itools.xhtml` includes:

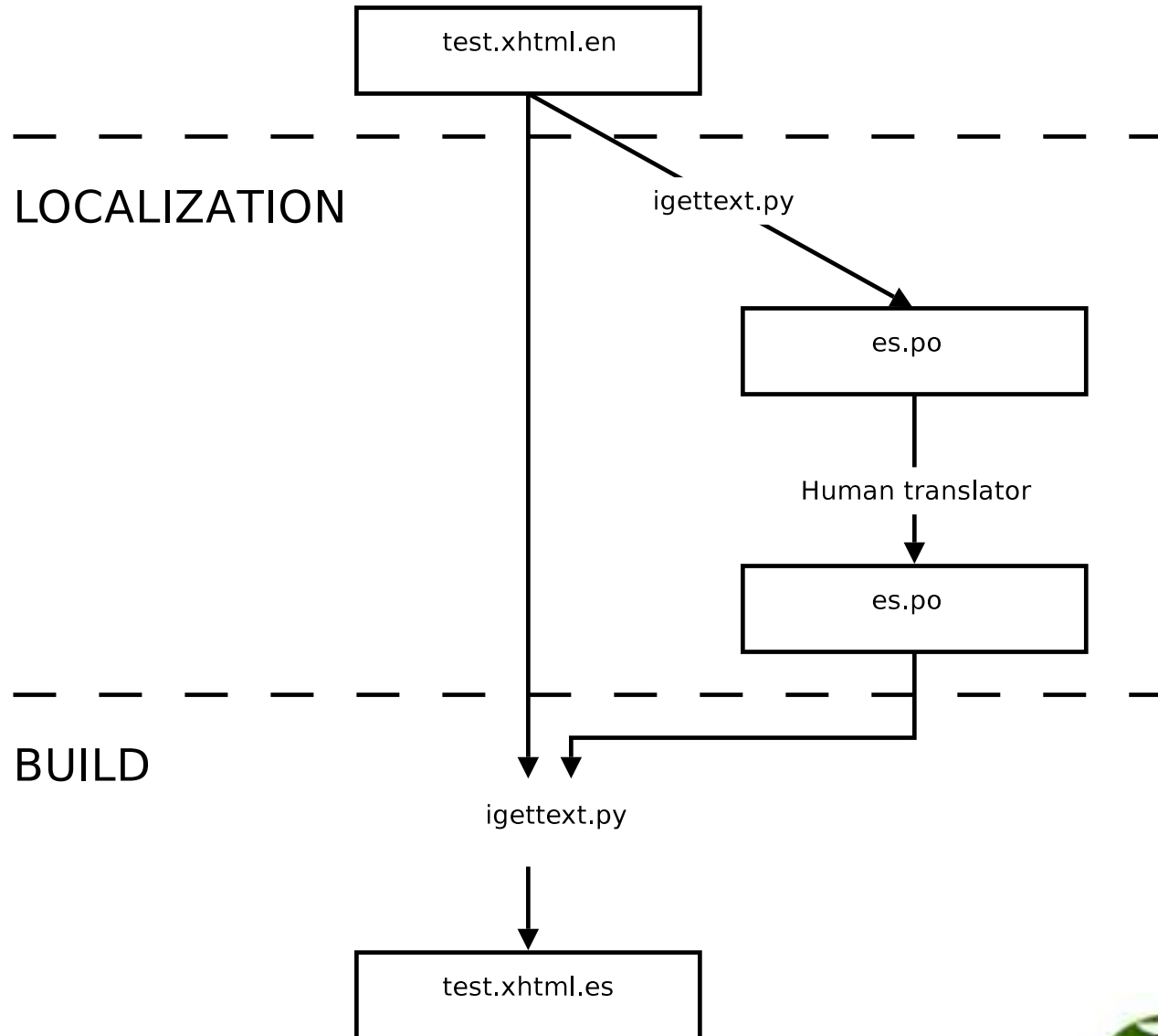
- A namespace handler for the XHTML namespace (<http://www.w3.org/1999/xhtml>).
- A resource handler class for XHTML documents.

The handler class `XHTML.Document` extends the base `XML.Document` class with an API specific to XHTML files.

Most notably some facilities to translate them.



# Translating XHTML Documents



# Translating XHTML Documents

The translation process is split in three steps:

1. Automatic extraction of translatable messages.

```
$ iggettext.py --pot test.xhtml.en > locale.pot  
$ iggettext.py --po locale.pot es.po > es.po
```

2. Human translation of the generated PO files.

3. Automatic building of the translated templates.

```
$ iggettext.py --xhtml test.xhtml.en es.po \  
> test.xhtml.es
```



# Translating XHTML Documents

The script `igettext.py` just loads the handlers for the XHTML and PO files, and uses the API they provide to do the hard work.



# Translating XHTML Documents

And what about the internationalization (i18n) step?

Reminder:

*Internationalization* is the operation by which a program, or a set of programs turned into a package, is made aware of and able to support multiple languages.

The good news are: *There is not any i18n step!*

The only rule is to write correct XHTML; in particular, to use block elements as block elements, and inline elements as inline elements.



# Translating XHTML Documents

This differs from traditional approaches. For example, the document below is a good source for `igettext.py`:

```
<h1>If</h1>
```

```
<p>
```

```
If you can make one heap of all your winnings  
And risk it on one turn of pitch-and-toss,  
And lose, and start again at your beginnings  
And never breathe a word about your loss;
```

```
</p>
```



# Translating XHTML Documents

But, if you do it the ZPT way (for instance), you will have to add explicit mark-up:

```
<h1 i18n:translate="">If</h1>
```

```
<p i18n:translate="">
```

```
If you can make one heap of all your winnings  
And risk it on one turn of pitch-and-toss,  
And lose, and start again at your beginnings  
And never breathe a word about your loss;  
</p>
```



# Translating XHTML Documents

Our technique has some advantages:

- Reduces the development costs by getting off the burden of internationalization from the developer's shoulders.  
Also, experience has proven that adding explicit mark-up is a tedious and very error-prone task.
- Reduces the performance overhead of making an application multilingual. Because it removes the need to do a lookup call for every sentence.
- And, this very same technique, and this very same code, is used not only for the user interfaces, but also for the content.



# Future developments



# Future developments

- A native XML parser implementing the very same programming interface.
- Support for XML based formats, including:
  - The *Translation Memory eXchange* (TMX) format.
  - The *XML Localisation Interchange File Format* (XLIFF).
- Keep improving the API and documentation.

# References

# References

- Web Site, <http://www.ikaaro.org/itools>
- Mailing list, <http://in-girum.net/mailman/listinfo/ikaaro>
- Contact *J. David Ibáñez*, [jdavid@itaapy.com](mailto:jdavid@itaapy.com)